Especialização em Engenharia de Software

Programação Orientada a Objetos

Exceções

Sérgio Soares

scbs@cin.ufpe.br

Classe de Contas: Definição

```
public class Conta {
   private String numero;
   private double saldo;
   ...
   public void debitar(double valor) {
      saldo = saldo - valor;
   }
}
```

Como evitar débitos acima do limite permitido?

Desconsiderar Operação

```
public class Conta {
   private String numero;
   private double saldo;
   ...
   public void debitar(double valor) {
      if (valor <= saldo)
         saldo = saldo - valor;
   }
}
```

Desconsiderar Operação

- Problemas:
  - quem solicita a operação não tem como saber se ela foi realizada ou não
  - nenhuma informação é dada ao usuário do sistema

Mostrar Mensagem de Erro

```
public class Conta {
  private String numero;
  private double saldo;
  ...
  public void debitar(double valor) {
    if (valor <= saldo)
      saldo = saldo - valor;
    else
      System.out.print("Saldo Insuficiente");
  }
}
```

Mostrar Mensagem de Erro

- Problemas:
  - informação é dada ao usuário do sistema, mas nenhuma sinalização de erro é fornecida para métodos que invocam **`debitar`**
  - supõe que a interface com o usuário é modo texto
  - cria uma forte dependência entre a classe **`Conta`** e sua interface com o usuário
  - entrelaçamento entre do código da camada de negócio com o código da camada de interface com o usuário

Retornar Código de Erro

```
public class Conta {
  private String numero;
  private double saldo;
  ...
  public boolean debitar(double valor) {
    boolean r  = false;
    if (valor <= saldo) {
      saldo = saldo - valor;
      r  = true;
    } else r = false;
    return r;
  }
}
```

Retornar Código de Erro

- Problemas:
  - dificulta a definição e o uso do método
  - não torna a causa ou o tipo do erro explícito
  - métodos que invocam **`debitar`** têm que testar o resultado retornado para decidir o que deve ser feito
  - A dificuldade é maior para métodos que retornam valores:
    - e se **`debitar`** já retornasse um outro valor qualquer? O que teria que ser feito?

Código de Erro: Problemas

```
public class CadastroContas {
   ...
   public int debitar(String numero,
                      double valor) {
      int erro = 0;
      Conta c = contas.procurar(numero);
      if (c != null) {
         boolean b = c.debitar(valor);
         if (b) erro = 0;
         else erro = 2;
      } else erro = 1;
      return erro;
   }
}
```

Exceções

- Ao invés de códigos, teremos exceções...
- São objetos comuns, portanto têm que ter uma classe associada
- Classes representando exceções herdam e são subclasses de **`Exception`** (pré-definida)
- Define-se subclasses de **`Exception`** para
  - oferecer informações extras sobre a falha, ou
  - distinguir os vários tipos de falhas

Exemplos de ocorrência de Exceções

- Acesso a um local do array fora dos limites
- Estouro aritmético
- Divisão por zero
- Esgotamento de memória
- Cast inválido
- Conta não encontrada no repositório
- Saldo insuficiente

As duas últimas são exceções específicas de aplicações e precisam ser definidas explicitamente

Definindo Exceções

```
public class SIException extends Exception {
  private double saldo;
  private String numero;
  public SIException(double saldo,
                     String numero) {
    super ("Saldo Insuficiente!");
    this.saldo = saldo;
    this.numero = numero;
  }
  public double getSaldo() {
    return saldo;
  }
  ...
}
```

Definindo Exceções (2)

```
public class CNEException extends Exception {
   public CNEException() {
      super (”Conta não encontrada");
   }
}
```

Definindo Métodos com Exceções (1)

```
public class Conta {
   ...
   public void debitar(double valor)
                          throws SIException {
      if (valor <= saldo)
         saldo = saldo - valor;
      else {
         SIException e;
         e = new SIException(saldo,numero);
         throw e;
      }
   }
}
```

Definindo Métodos com Exceções (2)

```
public class RepositorioContasArray
             implements RepositorioContas {
  private Conta[] contas;
  // ...
  public Conta procurar(String numero)
                          throws CNEException {
    Conta c = null;
    for (int i=0; i<indice && !acho; i++) {
      //...
    }
    if (!achou)
      throw new CNEException();
    return c;
  }
}
```

Definindo Métodos com Exceções (3)

```
public class Banco {
  private RepositorioContas contas;
  // ...
  public int debitar(String n, double v)
                throws SIException, CNEException {
    Conta c = contas.procurar(n);
    c.debitar(v);
  }
}
```

Definindo e Levantando Exceções

- **`Res metodo(Pars) throws E1,...,EN`**
- Todos os tipos de exceções levantadas no corpo de um método devem ser declaradas na sua assinatura
- Levantando exceções: **`throw`** *obj-exceção*
- Fluxo de controle e exceção são passados para a chamada do código que contém o comando **`throw`**, e assim por diante...

Já sabemos gerar exceções

- Mas onde e como tratar as mesmas?
  - O que fazer quando um erro acontece?

Tratando Exceções

```
Antes...
try {
   ...
   banco.debitar("123-4",90.00);
   ...
} catch (SIException e) {
   System.out.print(e.getMessage());
   System.out.print(" Conta/saldo: ");
   System.out.print(e.getNumero()+
                    " / " + e.getSaldo());
} catch (CNEException e) {...}
Depois...
```

Tratando Exceções

- A execução do **`try`** termina assim que uma exceção é levantada
- O *primeiro* **`catch`** de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último **`catch`**
- Se não houver nenhum **`catch`** compatível, a exceção e o fluxo de controle são passados para a chamada do código (método) com o **`try/catch`**

Tratando Exceções: Forma Geral

```
try {
   ...
} catch (E1 e1) {
   ...
}
...
} catch (En en) {
   ...
} finally {
   ...
}
```

O bloco finally é opcional desde que exista pelo menos um bloco catch

Tratando Exceções

- O bloco **`finally`** é sempre executado mesmo
  - após a terminação normal do **`try`**
  - após a execução de um **`catch`**
  - quando não existe nenhum **`catch`** compatível
- Quando o **`try`** termina normalmente ou um **`catch`** é executado, o fluxo de controle é passado para o código seguindo o bloco **`finally`** (depois deste ser executado)

Tratamento de Exceções

- Verificação de Exceções
  - Não verificadas
    - **`RuntimeException`**
    - **`NullPointerException`**
    - **`NumberFormatException`**
  - Verificadas
    - O compilador emite uma **mensagem de erro** indicando que a exceção **deve ser capturada**

Interfaces e exceções

- Uma interface deve declarar na assinatura dos seus métodos quais exceções (erros) os métodos (serviços) podem levantar (gerar)

- Implementações da interface (classes) podem ser mais eficientes e gerar menos erros do que os esperados

  - devem ser sempre um subconjunto das exceções definidas na interface

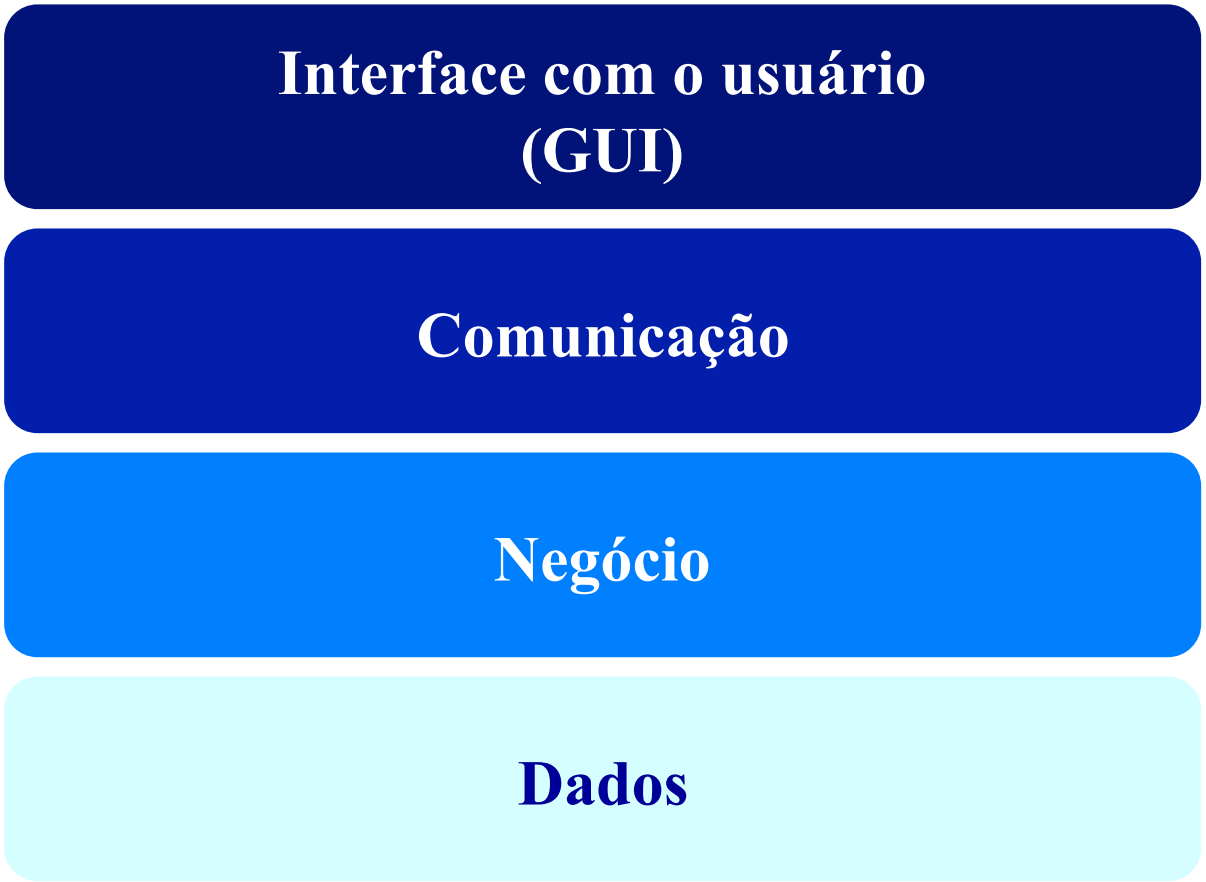
O MESMO VALE PARA MÉTODOS DE CLASSES ABSTRATAS

Exemplo

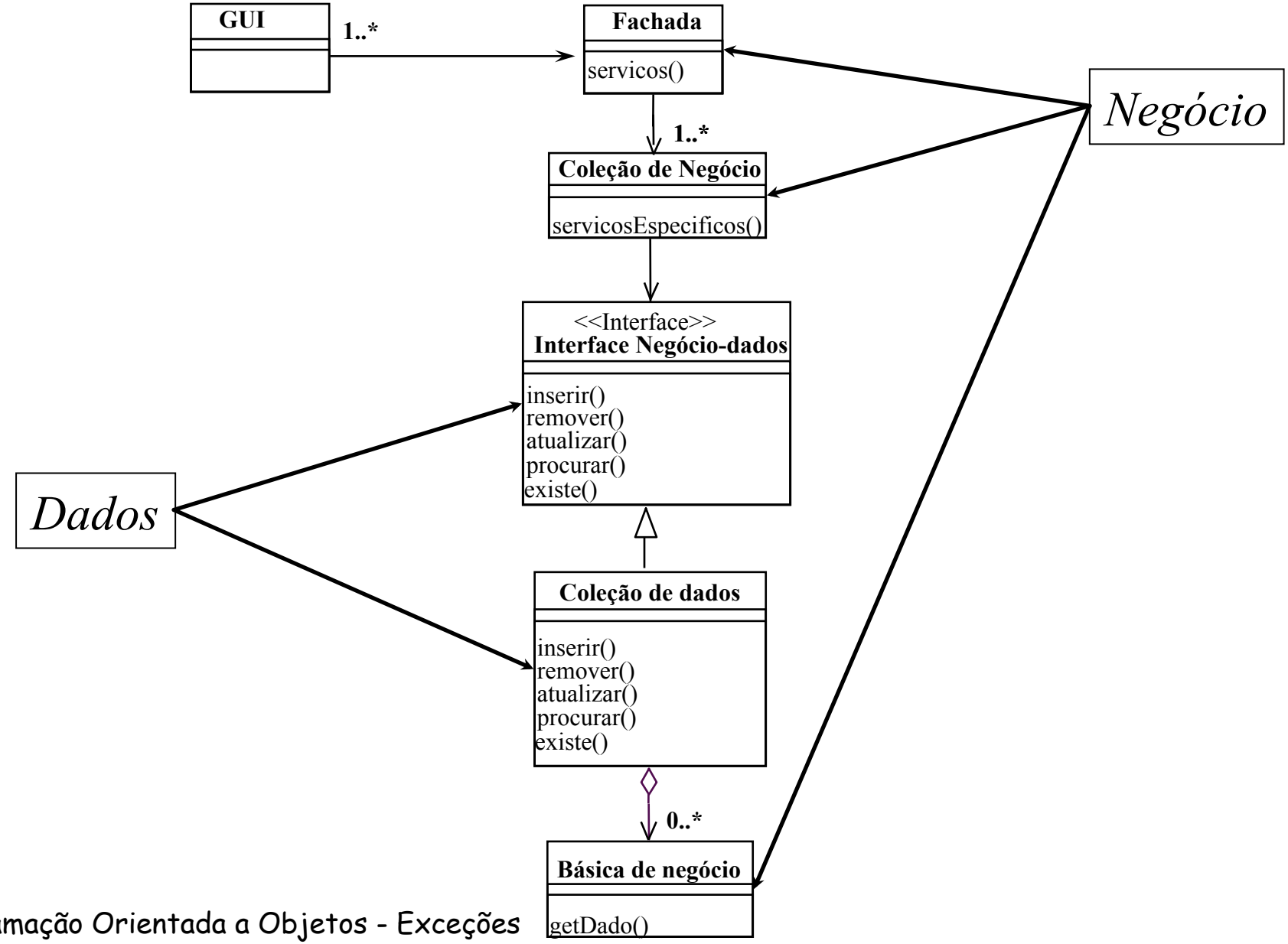
```
public interface RepositorioContas {
  void inserir(ContaAbstrata conta)
       throws RepositorioException;
  // ...
}
```

```
public class RepositorioContasArray
          implements RepositorioContas {
  public void inserir(ContaAbstrata conta){
    // nunca dará erro se o array encher
  }
  // ...
}
```

Vendo o código como um bolo... com várias camadas!



Arquitetura de software



Classes Básicas de Negócio

```
public class Conta {
   private double saldo;
   private String numero;
   private Cliente correntista;
   ...
   public void creditar(double valor) {
      saldo = saldo + valor;
   }
}
```

Cliente, Livro, Animal, Veiculo

Interfaces Negócio-Dados

```
public interface RepositorioContas {
  public void inserir(Conta conta)
      throws RepositorioException;
  public void atualizar(Conta conta)
      throws RepositorioException,
             ContaNaoEncontradaException;
  public void remover(String numero);
      throws RepositorioException,
             ContaNaoEncontradaException;
  ...
}
```

Classes Coleção de Dados

```
public class RepositorioContasArray
      implements RepositorioContas {
  ...
  public void inserir(Conta conta)
            throws RepositorioException{
    if (contas.length == indice)
      throw new RepositorioException(...);
    contas[indice] = conta;
    indice = indice + 1;
  } ...
}
```

Classes Coleção de Dados Persistentes

```
public class RepositorioContasBDR
      implements RepositorioContas {
  ...
  public void inserir(Conta conta)
      throws RepositorioException {
    try {
      ...
    } catch(SQLException ex) {
      throw new RepositorioException(ex);
    }
  } ...
}
```

Classes Coleção de Negócio

```
public class CadastroContas {
  private RepositorioContas contas;
  ...
  public void cadastrar(Conta conta)
      throws RepositorioException,
             ContaJaCadastradaException {
    if (!contas.existe(conta.getNumero())) {
      contas.inserir(conta);
    }
    else throw new ContaJaCadastradaException();
  } ...
}
```

Classe Fachada

```
public class Banco {
  private CadastroClientes clientes;
  public void cadastrar(Conta conta)
      throws RepositorioException,
             ContaJaCadastradaException
             CorrentistaNaoCadastradoException {
    Cliente c = conta.getCorrentista();
    if (clientes.existe(c.getCodigo())
      contas.cadastrar(conta);
    else
      throw new CorrentistaNaoCadastradoException();
  } ...
}
```

Especialização em Engenharia de Software

Programação Orientada a Objetos

Exceções

Sérgio Soares

scbs@cin.ufpe.br